



Ressourcesinformatiques

 + QUIZ

Version en ligne
OFFERTE !
pendant 1 an

Apprendre la Programmation Orientée Objet avec le langage Python

En téléchargement



corrigés des exercices



code source des exemples

2^e édition

Vincent BOUCHENY





Les exemples à télécharger sont disponibles à l'adresse suivante :

<http://www.editions-eni.fr>

Saisissez la référence ENI de l'ouvrage **RI2PYAPOO** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1

L'émergence de la POO

1. Ancêtres de la POO. 9
2. Besoin d'un langage de plus haut niveau. 11

Chapitre 2

Les concepts de la POO

1. Modélisation. 13
2. Objet et classe. 16
3. Encapsulation. 18
4. Agrégation et composition. 22
 - 4.1 Agrégation. 22
 - 4.2 Composition. 23
5. Interface. 23
6. Énumération. 26
7. Héritage. 27
 - 7.1 Héritage simple. 27
 - 7.2 Classe abstraite. 29
 - 7.3 Héritage multiple. 31
 - 7.4 Du « bon » usage de l'héritage. 33

2 — Apprendre la POO

avec le langage Python

8. Diagramme UML	36
8.1 Structure vs comportement	36
8.2 Diagramme de cas d'utilisation	36
8.3 Diagramme de séquence	38
9. Exercices corrigés	40
9.1 Classe simple	40
9.2 Relations entre classes	41
9.3 Héritage ?	43
9.4 Repérage des classes	45
9.5 Contenants et contenus	46
9.6 Membres	52

Chapitre 3

Présentation de l'environnement Python

1. Python, troisième du nom ?	57
2. Installation	58
2.1 python.org	58
2.2 Windows	59
2.3 Mac OS X	64
2.4 Unix/Linux	65
3. Outillage	66
3.1 pip	66
3.2 IDLE	67
3.3 PyCharm	68
4. Quelques concepts de base de Python	69
4.1 Introduction	69
4.2 Philosophie	69
4.3 Langage interprété et compilé	70
4.4 Duck typing	71
4.5 Modules	72

Chapitre 4

Les concepts de la POO avec Python

1. Classe.....	75
1.1 Déclaration	75
1.2 Instance.....	77
1.3 Membres d'une classe.....	79
1.3.1 Attribut	79
1.3.2 Méthode	82
1.4 Constructeur.....	86
1.5 Destructeur.....	89
1.6 Exercices	91
1.6.1 Palindrome - méthode de classe.....	91
1.6.2 Palindrome - méthode d'instance	92
1.6.3 Puzzle	93
1.6.4 Logger	95
2. Héritage.....	96
2.1 Construction.....	96
2.2 Polymorphisme.....	101
2.3 Héritage multiple	103
2.4 Exercices	109
2.4.1 Héritage « simple »	109
2.4.2 Puzzle	111
2.4.3 Héritage multiple - Diamant et paramètres de constructeur.....	114
2.4.4 Héritage multiple - Cas « réel »	116
3. Agrégation et composition.....	120
3.1 Agrégation.....	120
3.2 Composition.....	122
3.3 Exercices	124
3.3.1 Le jour d'après.....	124
3.3.2 Immortel ?.....	127
3.3.3 Alternative à l'héritage multiple	128

4 --- Apprendre la POO

avec le langage Python

4. Exception	132
4.1 Levée	132
4.2 Rattrapage	136
4.3 Éviter le masquage d'exception	140
4.4 Exception personnalisée	142
4.5 Exercice	144
5. Concepts de la POO non natifs	147
5.1 Classe abstraite	147
5.2 Interface	150
5.3 Encapsulation	151
6. Énumération	152
7. Duck typing	154

Chapitre 5

Un aperçu de quelques design patterns

1. Introduction	157
2. Singleton	159
3. Visiteur	166
3.1 Présentation	166
3.2 Exercice	172
4. Modèle - Vue - Contrôleur (MVC)	175
4.1 Présentation	175
4.2 Exercice	179
5. Abstract Factory	182
5.1 Présentation	182
5.2 Exercice	186

Chapitre 6 Plus loin avec Python

1. Introduction	189
2. XML.....	190
2.1 Présentation	190
2.2 DOM.....	191
2.2.1 Lecture.....	191
2.2.2 Méthode par accès direct	192
2.2.3 Méthode par analyse hiérarchique	192
2.2.4 Écriture	194
2.3 SAX.....	196
2.3.1 Lecture.....	196
2.3.2 Écriture	198
3. JSON	199
4. IHM.....	202
4.1 Tkinter	202
4.1.1 Création d'une fenêtre	202
4.1.2 Ajout de widgets.....	204
4.1.3 Gestion des événements.....	206
4.2 Qt.....	208
4.2.1 Présentation	208
4.2.2 Création d'une fenêtre	208
4.2.3 Ajout de widgets.....	210
4.2.4 Gestion des événements.....	212
5. Bases de données.....	214
5.1 Présentation	214
5.2 SQLite	216
6. Multithreading	221
6.1 Présentation	221
6.2 Python et la programmation concurrente.....	223
6.3 Utilisation du module threading.....	224
6.4 Synchronisation	227

6 — Apprendre la POO

avec le langage Python

6.5	Interblocage	229
7.	Développement web	231
7.1	Présentation	231
7.2	Création d'un projet Django	232
7.3	Développement web MVC	235
7.4	Quelques utilitaires intéressants	242

Chapitre 7

Quelques bonnes pratiques

1.	Introduction	245
2.	S'assurer avec des bases solides	246
2.1	De l'importance du socle	246
2.2	Structures de données et algorithmes communs	246
2.3	Un problème, plusieurs solutions	248
2.4	Choisir et maîtriser le bon outil	249
3.	Rester concis et simple	250
3.1	DRY	250
3.2	KISS	252
4.	Harmoniser l'équipe	253
4.1	Rédiger des conventions d'écriture	254
4.2	Revoir le code, tous ensemble	255
4.3	Documenter	256
4.4	Tester	256
4.5	Installer un environnement de déploiement continu	257
5.	Rejoindre une communauté	258
5.1	Participer à des conférences et groupes utilisateurs	259
5.2	Écouter la sagesse des foules	259
6.	Maîtriser les problèmes récurrents	261
6.1	Débogage	261
6.2	Traces	262
6.3	Monitoring	262

6.4	Performance	263
6.5	ETL	264
6.6	Bases de données : relationnelles ou non	265
6.7	Bases de données et ORM	266
6.8	Intégrations	267
6.9	Autres environnements logiciels	268
7.	Poursuivre sa croissance personnelle	268
7.1	Savoir poser des questions	269
7.2	Open source	270
7.3	Architecture	271
8.	Poursuivre sa croissance professionnelle	271
8.1	L'humain	271
8.2	Développement rapide et itérations	272
9.	Conclusion	272
	Index	273

Chapitre 4

Les concepts de la POO avec Python

1. Classe

1.1 Déclaration

Une classe est la définition d'un concept métier, elle contient des attributs (des valeurs) et des méthodes (des fonctions).

En Python, le nom d'une classe ne peut commencer par un chiffre ou un symbole de ponctuation, et ne peut pas être un mot-clé du langage comme `while` ou `if`. À part ces contraintes, Python est très permissif sur le nom des classes et variables en autorisant même les caractères accentués. Cette pratique est cependant extrêmement déconseillée à cause des problèmes de compatibilité entre différents systèmes.

Voici l'implémentation d'une classe en Python ne possédant aucun membre : ni attribut, ni méthode.

```
class MaClass:
    # Pour l'instant, la classe est déclarée vide,
    # d'où l'utilisation du mot-clé 'pass'.
    pass
```

Le mot-clé `class` est précédé du nom de la classe. Le corps de la classe est lui indenté comme le serait le corps d'une fonction. Dans un corps de classe, il est possible de définir :

- des fonctions (qui deviendront des méthodes de la classe) ;
- des variables (qui deviendront des attributs de la classe) ;
- des classes imbriquées, internes à la classe principale.

Les méthodes et les attributs seront présentés dans les sections suivantes éponymes.

Il est possible d'organiser le code de façon encore plus précise grâce à l'imbriication de classes. Tout comme il est possible, voire conseillé, de répartir les classes d'une application dans plusieurs fichiers (qu'on appelle « modules » en Python), il peut être bien plus lisible et logique de déclarer certaines classes dans une classe « hôte ». La déclaration d'une classe imbriquée dans une autre ressemble à ceci :

```
# Classe contenante, déclarée normalement.
class Humain :

    # Classes contenues, déclarées normalement aussi,
    # mais dans le corps de la classe contenante.

    class Femme :
        pass

    class Homme:
        pass
```

La seule implication de l'imbriication est qu'il faut désormais passer par la classe `Humain` pour utiliser les classes `Femme` et `Homme` en utilisant l'opérateur d'accès « point » : `Humain.Femme` et `Humain.Homme`. Exactement comme on le ferait pour un membre classique.

L'imbriication n'impacte en rien le comportement du contenant ou du contenu.

1.2 Instance

Si l'on exécute le code précédent de déclaration de classe vide, rien ne s'affiche. Ceci est normal puisque l'on n'a fait que déclarer une classe. Après cette exécution, l'environnement Python sait qu'il existe désormais une classe nommée **MaClasse**. Maintenant, il va falloir l'utiliser.

Pour rappel : une classe est une définition, une abstraction de l'esprit. C'est elle qui permet de présenter, d'exposer les données du concept qu'elle représente. Afin de manipuler réellement ces données, il va falloir une représentation concrète de cette définition. C'est l'instance, ou l'objet, de la classe.

Une instance (ou un objet) est un exemplaire d'une classe. L'instanciation, c'est le mécanisme qui permet de créer un objet à partir d'une classe. Si l'on compare une instance à un vêtement, alors la classe est le patron ayant permis de le découper, et la découpe en elle-même est l'instanciation. Par conséquent, une classe peut générer de multiples instances, mais une instance ne peut avoir comme origine qu'une seule classe.

Ainsi donc, si l'on veut créer une instance de **MaClasse** :

```
■ instance = MaClasse()
```

Les parenthèses sont importantes : elles indiquent un appel de méthode. Dans le cas précis d'une instanciation de classe, la méthode s'appelle `__init__` : c'est le constructeur de la classe. Si cette méthode n'est pas implémentée dans la classe, alors un constructeur par défaut est automatiquement appelé, comme c'est le cas dans cet exemple.

Il est désormais possible d'afficher quelques informations sur la console :

```
■ print(instance)
>>> <__main__.MaClasse object at 0x10f039f60>
```

Lorsqu'on affiche sur la sortie standard la variable `instance`, l'interpréteur informe qu'il s'agit d'un objet de type `__main__.MaClasse` dont l'adresse mémoire est `0x10f039f60`.

D'où vient ce `__main__` ? Il s'agit du module dans lequel **MaClasse** a été déclarée. En Python, un fichier source correspond à un module, et le fichier qui sert de point d'entrée à l'interpréteur est appelé `__main__`. Le nom du module est accessible via la variable spéciale `__name__`.

```
# Ceci est le module b.
print("Nom du module du fichier b.py : " + __name__)
chiffre = 42
```

b.py

```
# Ceci est le module a.
print("Nom du module du fichier a.py : " + __name__)
import b
print(b.chiffre)
```

a.py

```
$> python a.py
Nom du module du fichier a.py : __main__
Nom du module du fichier b.py : b
42
```

sortie standard

Le fichier **a.py** sert d'entrée à l'interpréteur Python : ce module se voit donc attribuer **__main__** comme nom. Lors de l'import du module **b.py**, le fichier est entièrement lu et affiche le nom du module qui correspond, lui, au nom du fichier.

Une classe faisant toujours partie d'un module, la classe **MaClasse** a donc été assignée au module **__main__**.

L'adresse hexadécimale de la variable **instance** correspond à l'emplacement mémoire réservé pour stocker cette variable. Cette adresse permet, entre autres, de différencier deux variables qui pourraient avoir la même valeur.

```
a = MaClasse()
print("Info sur 'a' : {}".format(a))
>>> Info sur 'a' : <__main__.MaClasse object at 0x10b61f908>

b = a
print("Info sur 'b' : {}".format(b))
>>> Info sur 'b' : <__main__.MaClasse object at 0x10b61f908>

b = MaClasse()
print("Info sur 'b' : {}".format(b))
>>> Info sur 'b' : <__main__.MaClasse object at 0x10b6797b8>
```

L'adresse de **a** est 0x10b61f908. Lorsqu'on assigne **a** à **b** et qu'on affiche **b**, on constate que les variables pointent vers la même zone mémoire. Par contre, si l'on assigne à **b** une nouvelle instance de **MaClasse**, alors son adresse n'est plus la même : il s'agit d'une nouvelle zone mémoire allouée pour stocker un nouvel exemplaire de **MaClasse**.

Voici tout ce qu'il y a à savoir sur la variable **instance**. D'autres informations sont accessibles concernant la classe elle-même :

```
print(MaClasse)
>>> <class '__main__.MaClasse'>

classe = MaClasse
print(classe)
>>> <class '__main__.MaClasse'>
```

En Python, tout est objet, y compris les classes. N'importe quel objet peut être affecté à une variable, et les classes ne font pas exception. Il est donc tout à fait valide d'assigner **MaClasse** à une variable **classe**, et son affichage sur la sortie standard confirme bien que **classe** est une classe, et pas une instance. D'où l'importance des parenthèses lorsqu'on désire effectuer une instanciation de classe. En effet, les parenthèses précisent bien qu'on appelle le constructeur de la classe, et on obtient par conséquent une instance. L'omission des parenthèses signifie que l'on désigne la classe elle-même.

1.3 Membres d'une classe

1.3.1 Attribut

Un attribut est une variable associée à une classe. Pour définir un attribut au sein d'une classe, il suffit :

– d'assigner une valeur à cet attribut dans le corps de la classe :

```
class Cercle:
    # Déclaration d'un attribut de classe 'rayon'
    # auquel on assigne la valeur 2.
    rayon = 2

print(Cercle.rayon)
>>> 2
```

- d'assigner une valeur à cet attribut en dehors de la classe. On parle alors d'attribut dynamique :

```
class Cercle:
    # Le corps de la classe est laissé vide.
    pass

# Déclaration dynamique d'un attribut de classe 'rayon'
# auquel on assigne la valeur 2.
Cercle.rayon = 2

print(Cercle.rayon)
>>> 2
```

Les attributs définis ainsi sont appelés « attributs de classe » car ils sont liés à la classe, par opposition aux « attributs d'instance » dont la vie est liée à l'instance à laquelle ils sont rattachés. Les attributs de classe sont automatiquement reportés dans les instances de cette classe et deviennent des attributs d'instance.

```
c = Cercle()
print(c.rayon)
>>> 2
```

Puisqu'un attribut de classe est lié à la classe, et non pas à l'instance, il existe durant toute la durée d'exécution du programme. Plus précisément, il existe tant que la classe à laquelle il est lié est définie. Un attribut d'instance, quant à lui, n'existe qu'à travers l'instance qui lui est liée. Ainsi, si l'instance est détruite, l'attribut d'instance l'est également.

```
# Déclaration d'une instance de Cercle.
c = Cercle()
# Déclaration d'un attribut d'instance 'rayon'.
c.rayon = 5
# Affichage de cet attribut d'instance.
print(c.rayon)
>>> 5

# Destruction de l'instance de Cercle.
del(c)
# Affichage de l'attribut d'instance.
print(c.rayon)
>>> Traceback (most recent call last):
```

```
File "a.py", line 28, in <module>
    print(c.rayon)
NameError: name 'c' is not defined
# c n'est plus défini dans l'environnement d'exécution Python.
# Par conséquent, les attributs liés à cette instance non plus.
```

Tandis qu'un attribut de classe survit à toutes les instances de cette classe.

```
# Déclaration d'une instance de Cercle.
c = Cercle()
# Destruction de l'instance de Cercle.
del(c)
# Affichage de l'attribut de classe.
print(Cercle.rayon)
>>> 2
# L'attribut de classe existe toujours.
```

Si l'attribut de classe est copié dans chaque objet instancié en tant qu'attribut d'instance, il n'en demeure pas moins que ces attributs demeurent indépendants l'un de l'autre. Toute modification sur l'attribut d'instance n'a aucun impact sur l'attribut de classe. De façon similaire, si la valeur de l'attribut de classe est changée, cela n'a aucun impact sur les objets déjà instanciés (mais cela en aura évidemment sur les futures instances) :

```
c.rayon = 4
print(c.rayon)
>>> 4
# Attribut de l'instance c.

print(Cercle.rayon)
>>> 2
# Attribut de la classe Cercle.

Cercle.rayon = 6
print(Cercle.rayon)
>>> 6
# Attribut de la classe Cercle
# dont la valeur vient d'être modifiée.

print(c.rayon)
>>> 4
# Attribut de l'instance c, qui demeure inchangé.
```